CHAPTER 2
Exploratory Data Analysis

Introduction

Exploratory Data Analysis (EDA) is the technique of examining, understanding, and summarizing data using various methods. EDA uncovers important insights, features, characteristics, patterns, relationships, and outliers. It also generates hypotheses for the research questions and covers descriptive statistics, a graphical representation of data in a meaningful way, and data exploration in general. In this chapter, we present techniques for data aggregation, transformation, normalization, standardization, binning, grouping, data coding, and encoding, handling missing data and outliers, and the appropriate data visualization methods.

Structure

In this chapter, we will discuss the following topics:

- Exploratory data analysis and its importance
- Data aggregation
- Data normalization, standardization, and transformation
- Data binning, grouping, encoding

- Missing data, detecting and treating outliers
- Visualization and plotting of data

Objectives

By the end of this chapter, readers will learn the techniques to explore the data and to gather meaningful insight to know data well. You will acquire the skills necessary to explore data and gain insights for better understanding. You will learn different data preprocessing method and how to apply them. Further this chapter also explains data encoding, grouping, cleansing, and visualization techniques with Python.

Exploratory data analysis and its importance

EDA is a method of analyzing and summarizing data sets to discover their key characteristics, often using data visualization techniques. EDA helps you better understand the data, find patterns and outliers, test hypotheses, and check assumptions. For example, if you have a data set of home prices and characteristics, you can use EDA to explore the distribution of prices, the relationship between price and characteristics, the effect of location and neighborhood, and so on. You can also use EDA to check for missing values, outliers, or errors in the data. In data science and analytics, EDA helps prepare data for further analysis and modeling. It can help select the appropriate statistical methods or machine learning algorithms for the data, validate the results, and communicate the findings.

Python is a popular programming language for EDA, as it has many libraries and tools that support data manipulation, visualization, and computation. Some of the commonly used libraries for EDA in Python are **pandas**, **NumPy**, **Matplotlib**, **Seaborn**, **Statsmodels**, **Scipy** and **Scikit-learn**. These libraries provide functions and methods for

reading, cleaning, transforming, exploring, and visualizing data in various formats and dimensions.

Data aggregation

Data aggregation in statistics involves summarizing numerical data using statistical measures like mean, median, mode, standard deviation, or percentile. This approach helps detect irregularities and outliers, and enables effective analysis. For example, to determine the average height of students in a class, their individual heights can be aggregated using the mean function, resulting in a single value representing the central tendency of the data. To evaluate the extent of variation in student heights, utilize the standard deviation function to gather data, which will indicate how spread out the data is from the average. The practice of data aggregation in statistics can simplify and aid in comprehending large data sets.

Mean

The mean is a statistical measure used to determine the average value of a set of numbers. To obtain the mean, add all numbers and divide the sum by the number of values. For example, if you have five test scores: 80, 90, 70, 60, and 100, the mean will be as follows:

$$Mean = (80 + 90 + 70 + 60 + 100) / 5$$

The average score will be the typical score for this series of tests.

Tutorial 2.1: An example to compute the mean from a list of numbers, is as follows:

```
# Define a list of test scores
test_scores = [80, 90, 70, 60, 100]
# Calculate the sum of the test scores
total = sum(test_scores)
```

```
# Calculate the number of test scores
count = len(test_scores)
# Calculate the mean by dividing the sum by the count
mean = total / count
# Print the mean
print("The mean is", mean)
```

The Python **sum()** function takes a list of numbers and returns their sum. For instance, **sum([1, 2, 3])** equals 6. On the other hand, the **len()** function calculates the number of elements in a sequence like a string, a list, or a tuple. For example, **len("hello")** returns **5**.

Output:

```
The mean is 80.0
```

Median

Median determines the middle value of a data set by locating the value positioned at the center when the data is arranged from smallest to largest. When there is an even number of data points, the median is calculated as the average of the two middle values. For example, among test scores: 75, 80, 85, 90, 95. To determine the median, we must sort the data and locate the middle value. In this case the middle value is 85 thus, the median is 85. If we add another score of 100 to the dataset, we now have six data points: 75, 80, 85, 90, 95, 100. Therefore, the median is the average of the two middle values 85 and 90. The average of the two values: *(85 + 90) / 2 = 87.5*. Hence, the median is 87.5.

Tutorial 2.2: An example to compute the median is as follows:

```
# Define the dataset as a list
data = [75, 80, 85, 90, 95, 100]
# Calculate the number of data points
```

```
num_data_points = len(data)
# Sort the data in ascending order
data.sort()
# Check if the number of data points is odd
if num_data_points % 2 == 1:
    # If odd, find the middle value (median)
    median = data[num_data_points // 2]
else:
    # If even, calculate the average of the two middle values
    middle1 = data[num_data_points // 2 - 1]
    middle2 = data[num_data_points // 2]
    median = (middle1 + middle2) / 2
# Print the calculated median
print("The median is:", median)
```

Output:

```
The median is: 87.5
```

The median is a useful tool for summarizing data that is skewed or has outliers. It is more reliable than the mean, which can be impacted by extreme values. Furthermore, the median separates data into two equal quartiles.

Mode

Mode represents the value that appears most frequently in a given data set. For example, consider a set of shoe sizes that is, 6, 7, 7, 8, 8, 8, 9, 10. To find the mode, count how many times each value appears and identify the value that occurs most frequently. The mode is the most common value. In this case, the mode is 8 since it appears three times, more than any other value.

Tutorial 2.3: An example to compute the mode, is as follows:

```
# Define the dataset as a list
shoe_sizes = [6, 7, 7, 8, 8, 8, 9, 10]
# Create an empty dictionary to store the count of each value
size_counts = {}
# Iterate through the dataset to count occurrences
for size in shoe_sizes:
    if size in size_counts:
        size_counts[size] += 1
    else:
        size_counts[size] = 1
# Find the mode by finding the key with the maximum value in the dictionary
mode = max(size_counts, key=size_counts.get)
# Print the mode
print("The mode is:", mode)
```

max() used in *tutorial 2.3* is a Python function that returns the highest value from an iterable such as a list or dictionary. In this instance, it retrieves the key (**shoe_sizes**) with the highest count in the **size_counts** dictionary. The **.get()** method is used in a dictionary as a key function for **max()**. It retrieves the value associated with a key. In this case, **size_counts.get** retrieves the count associated with each shoe size key. Then **max()** uses this information to determine which key (**shoe_sizes**) has the highest count, indicating the mode.

Output:

```
The mode is: 8
```

Variance

Variance measures the deviation of data values from their average in a dataset. It is calculated by averaging the

squared differences between each value and the mean. A high variance suggests that data is spread out from the mean, while a low variance suggests that data is tightly grouped around the mean. For example, suppose we have two sets of test scores: **A = [90, 92, 94, 96, 98]** and **B = [70, 80, 90, 100, 130]**. The mean of both sets is 94, but the variance of A is 8 and B is 424. Lower variance in A means the scores in A are more consistent and closer to the mean than the scores in B. We can use the **var()** function from the **numpy** module to see the variance in Python.

Tutorial 2.4: An example to compute the variance is as follows:

```
1. import numpy as np
2. # Define two sets of test scores
3. A = [90, 92, 94, 96, 98]
4. B = [70, 80, 90, 100, 130]
5. # Calculate and print the mean of A and B
6. print("The mean of A is", sum(A)/len(A))
7. print("The mean of B is", sum(B)/len(B))
8. # Calculate and print the variance of A and B
9. var_A = np.var(A)
10. var_B = np.var(B)
11. print("The variance of A is", var_A)
12. print("The variance of B is", var_B)
```

To compute the variance in a pandas data frame, one way is to use the **describe()** method, which returns a summary of the descriptive statistics for each column, including the variance. For example, if we have a data frame named **df**, we can use **df.describe()** to see the variance of each column. Another way is to use the **apply()** method, which applies a function to each column or row of a data frame. For example, if we want to compute the variance of each row, we can use **df.apply(np.var, axis=1)**, where **np.var** is

the NumPy function for variance and **axis=1** means that the function is applied along the row axis.

Output:

```
The mean of A is 94.0
The mean of B is 94.0
The variance of A is 8.0
The variance of B is 424.0
```

Standard deviation

Standard deviation is a measure of how much the values in a data set vary from the mean. It is calculated by taking the square root of the variance. A high standard deviation means that the data is spread out, while a low standard deviation means that the data is concentrated around the mean. For example, suppose we have two sets of test scores: **A = [90, 92, 94, 96, 98]** and **B = [70, 80, 90, 100, 110]**. The mean of both sets is 94, but the standard deviation of A is about 2.83 and the standard deviation of B is about 14.14. This means that the scores in A are more consistent and closer to the mean than the scores in B. To find the standard deviation in Python, we can use the **std()** function from the **numpy** module.

Tutorial 2.5: An example to compute the standard deviation is as follows:

```
# Import numpy module
import numpy as np
# Define two sets of test scores
A = [90, 92, 94, 96, 98]
B = [70, 80, 90, 100, 110]
# Calculate and print the standard deviation of A and B
std_A = np.std(A)
std_B = np.std(B)
print("The standard deviation of A is", std_A)
```

```
10. print("The standard deviation of B is", std_B)
```

Output:

```
1. The standard deviation of A is 2.82
2. The standard deviation of B is 14.14
```

Quantiles

A quantile is a value that separates a data set into an equal number of groups, typically four (quartiles), five (quintiles), or ten (deciles). The groups are formed by ranking the data set in ascending order, ensuring that each group contains the same number of values. Quantiles are useful for summarizing data distribution and comparing different data sets.

For example, let us consider a set of 15 heights in centimeters: **[150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178]**. To calculate the quartiles (a specific subset of quantiles) for this dataset, divide it into four equally sized groups. Q1, the first quartile, represents the median of the lower half of the data, which is 158. Q2, the second quartile, corresponds to the median of the entire data set, which is 164. Q3, the third quartile, represents the median of the upper half of the data, which is 170. The data is split into four clear groups by the quartiles: **[150, 152, 154, 156], [158, 160, 162], [164, 166, 168], and [170, 172, 174, 176, 178]**. This separation facilitates understanding and comparison of distinct segments of the data's distribution.

Tutorial 2.6: An example to compute the quantiles is as follows:

```
1. # Import numpy module
2. import numpy as np
3. # Define a data set of heights in centimeters
4. heights = [150 ,152 ,154 ,156 ,158 ,160 ,162 ,164 ,166 ,
   168 ,170 ,172 ,174 ,176 ,178]
```

```
# Calculate and print the quartiles of the heights
Q1 = np.quantile(heights ,0.25)
Q2 = np.quantile(heights ,0.5)
Q3 = np.quantile(heights ,0.75)
print("The first quartile is", Q1)
print("The second quartile is", Q2)
print("The third quartile is", Q3)
```

Output:

```
The first quartile is 157.0
The second quartile is 164.0
The third quartile is 171.0
```

Tutorial 2.7: An example to compute mean, median, mode, standard deviation, maximum, minimum value in pandas data frame.

The mean, median, mode, variance, maximum and minimum value in data frame can be computed easily with **mean()**, **median()**, **mode()**, **var()**, **max()**, **min()** respectively, as follows:

```
# Import the pandas library
import pandas as pd
# Import display function
from IPython.display import display
# Load the diabetes data from a csv file
diabetes_df = pd.read_csv(
    "/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
# Print the mean of each column
print(f'Mean: \n {diabetes_df.mean()}')
# Print the median of each column
print(f'Median: \n {diabetes_df.median()}')
# Print the mode of each column
```

```
print(f'Mode: \n {diabetes_df.mode()}')
# Print the variance of each column
print(f'Varience: \n {diabetes_df.var()}')
# Print the standard deviation of each column
print(f'Standard Deviation: \n{diabetes_df.std()}')
# Print the maximum value of each column
print(f'Maximum: \n {diabetes_df.max()}')
# Print the minimum value of each column
print(f'Minimum: \n {diabetes_df.min()}')
```

Tutorial 2.8: An example to compute mean, median, mode, standard deviation, maximum, minimum value in NumPy array, is as follows:

```
# Import the numpy and statistics libraries
import numpy as np
import statistics as st
# Create a numpy array with some data
data = np.array([12, 15, 20, 25, 30, 30, 35, 40, 45, 50])
# Calculate the mean of the data using numpy
mean = np.mean(data)
# Calculate the median of the data using numpy
median = np.median(data)
# Calculate the mode of the data using statistics
mode_result = st.mode(data)
# Calculate the standard deviation of the data using num
py
std_dev = np.std(data)
# Find the maximum value of the data using numpy
maximum = np.max(data)
# Find the minimum value of the data using numpy
minimum = np.min(data)
# Print the results to the console
```

```
print("Mean:", mean)
print("Median:", median)
print("Mode:", mode_result)
print("Standard Deviation:", std_dev)
print("Maximum:", maximum)
print("Minimum:", minimum)
```

Output:

```
Mean: 30.2
Median: 30.0
Mode: 30
Standard Deviation: 11.93
Maximum: 50
Minimum: 12
```

Tutorial 2.9: An example to compute variance, quantiles, and percentiles using **var()** and **quantile** from diabetes dataset data frame, and also **describe()** to describe the data frame, is as follows:

```
import pandas as pd
from IPython.display import display
# Load the diabetes data from a csv file
diabetes_df = pd.read_csv(
    "/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
# Calculate the variance of each column using pandas
variance = diabetes_df.var()
# Calculate the quantiles (25th, 50th, and 75th percentiles) of each column using pandas
quantiles = diabetes_df.quantile([0.25, 0.5, 0.75])
# Calculate the percentiles (90th and 95th percentiles) of each column using pandas
percentiles = diabetes_df.quantile([0.9, 0.95])
# Display the results using the display function
```

```
13. display("Variance:", variance)
14. display("Quantiles:", quantiles)
15. display("Percentiles:", percentiles)
```

This will calculate the variance, quantile and percentile of each column in the **diabetes_df** data frame.

Data normalization, standardization, and transformation

Data normalization, standardization, and transformation are methods for preparing data for analysis. They ensure that the data is consistent, comparable, and appropriate for various analytical techniques. **Data normalization** rescales feature values to a range between zero and one, helping to mitigate the impact of outliers and different scales on the data. For instance, if one feature ranges from 0 to 100, while another ranges from 0 to 10,000, normalizing them can enhance comparability.

Standardizing data is achieved by subtracting the mean and dividing by the standard deviation of a feature. This results in a more normal distribution of data centered around zero. For example, if one feature has a mean of 50 and a standard deviation of 10, standardizing it will achieve a mean of 0 and a standard deviation of 1.

Data transformation involves using a mathematical function to alter the shape or distribution of a feature and make the data more linear or symmetrical. For instance, if a feature has an uneven distribution, applying a logarithmic or square root transformation can balance it. The order of these techniques relies on the data's purpose and type. It is generally recommended to perform data transformation prior to data standardization and then data normalization. Nevertheless, specific methods may call for varying or no preprocessing. Therefore, understanding the requirements and assumptions of each technique is crucial before

implementation.

Data normalization

Standardizing and organizing data entries through normalization improves their suitability for analysis and comparison, resulting in higher quality data. Additionally, reducing the impact of outliers enhances algorithm performance, increases data interpretability, and uncovers underlying patterns among variables.

Normalization of NumPy array

We can use the **numpy.min** and **numpy.max** functions to find the minimum and maximum values of an array, and then use the formula *xnorm = (xi – xmin) / (xmax – xmin)* to normalize each value.

Tutorial 2.10: An example to show normalization of NumPy array, is as follows:

```
#import numpy
import numpy as np
#create a sample dataset
data = np.array([10, 15, 20, 25, 30])
#find the minimum and maximum values of the data
xmin = np.min(data)
xmax = np.max(data)
#normalize the data using the formula
normalized_data = (data - xmin) / (xmax - xmin)
#print the normalized data
print(normalized_data)
```

Array data before normalization, is as follows:

```
[10 15 20 25 30]
```

Array data after normalization, is as follows:

```
[0.   0.25 0.5  0.75 1.  ]
```

Tutorial 2.11: An example to show normalization of the 2-Dimensional NumPy array using **MinMaxScalar,** is as follows:

Following is an easy example of data normalization in Python using the scikit-learn library. **MinMaxScaler** is a technique to rescale the values of a feature to a specified range, typically between zero and one. This can help to reduce the effect of outliers and different scales on the data. **scaler.fit_transform()** is a method that combines two steps: fit and transform. The fit step computes the minimum and maximum values of each feature in the data. The transform step applies the formula *xnorm = (xi – xmin) / (xmax – xmin)* to each value in the data, where *xmin* and *xmax* are the minimum and maximum values of the feature.

Code:

```
#import numpy library for working with arrays
import numpy as np
#import MinMaxScaler class from the preprocessing module of scikit-learn library for data normalization
from sklearn.preprocessing import MinMaxScaler
#create a structured data as a 2D array with two features: x and y
structured_data = np.array([[100, 200], [300, 400], [500, 600]])
#print the normalized structured data as a numpy array
print("Original Data:")
print(structured_data)
#create an instance of MinMaxScaler object that can normalize the data
scaler = MinMaxScaler()
#fit the scaler to the data and transform the data to a range between 0 and 1
normalized_structured = scaler.fit_transform(structured
```

```
_data)
#print the normalized structured data as a numpy array
print("Normalized Data:")
print(normalized_structured)
```

2-Dimensional array data before normalization is as follows:

```
[[100 200]
 [300 400]
 [500 600]]
```

2-Dimensional array data after normalization is as follows:

```
[[0.  0. ]
 [0.5 0.5]
 [1.  1. ]]
```

One potential problem when using **MinMaxScaler** for normalization is its sensitivity to outliers and extreme values. This can distort the scaling and limit the range of transformed features, potentially impacting the performance and accuracy of machine learning algorithms that rely on feature scale or distribution. A better alternative could be using the **Standard Scaler** or the **Robust Scaler**.

Standard Scaler rescales the data to achieve a mean of zero and a standard deviation of one, which improves optimization or distance-based algorithms. Although outliers can still impact the data, there is no guarantee of a restricted range for the transformed features. Robust Scaler is robust against extreme values and outliers, as it eliminates the median and rescales the data based on the **Interquartile Range (IQR)**. However, there is no assurance of a bounded span for the transformed features.

Tutorial 2.12. An example to show normalization of the 2-Dimensional array, is as follows:

```
#import the preprocessing module from the scikit-learn library
```

```
from sklearn import preprocessing
#create a sample dataset with two features: x and y
data = [[10, 2000], [15, 3000], [20, 4000], [25, 5000]]
#initialize a MinMaxScaler object that can normalize the data
scaler = preprocessing.MinMaxScaler()
#fit the scaler to the data and transform the data to a range between 0 and 1
normalized_data = scaler.fit_transform(data)
#print the normalized data as a numpy array
print(normalized_data)
```

The data before normalization, is as follows:

```
[[10, 2000], [15, 3000], [20, 4000], [25, 5000]]
```

The data after normalization represented between zero and one, is as follows:

```
[[0.         0.        ]
[0.33333333 0.33333333]
[0.66666667 0.66666667]
[1.         1.        ]]
```

Normalization of pandas data frame

To normalize a pandas data frame we can use the min-max scaling technique. Min-max scaling is a normalization method that rescales data to fit between zero and one. It is beneficial for variables with predetermined ranges or algorithms that are sensitive to scale. An example of min-max scaling can be seen by normalizing test scores that range from 0 to 100.

Following are some sample scores to consider:

Name	Score
Alice	80

Bob	60
Carol	90
David	40

***Table 2.1**: Scores of students in a class*

To apply min-max scaling, we use the following formula:

normalized value = (original value - minimum value) / (maximum value - minimum value)

The minimum value is 0 and the maximum value is 100, so we can simplify the formula as follows:

normalized value = original value / 100

Using this formula, we can calculate the normalized scores as follows:

Name	Score	Normalized score
Alice	80	0.8
Bob	60	0.6
Carol	90	0.9
David	40	0.4

***Table 2.2:** Normalized scores of students in a class*

The normalized scores are now between zero and one, and they preserve the relative order and distance of the original scores.

Tutorial 2.13. An example to show normalization of data frame using **pandas** and **sklearn** library, is as follows:

```
#import pandas and sklearn
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
#create a sample dataframe with three columns: age, height, and weight
df = pd.DataFrame({
```

```
    'age': [25, 35, 45, 55],
    'height': [160, 170, 180, 190],
    'weight': [60, 70, 80, 90]
})
#print the original dataframe
print("Original dataframe:")
print(df)
#create a MinMaxScaler object
scaler = MinMaxScaler()
#fit and transform the dataframe using the scaler
normalized_df = scaler.fit_transform(df)
#convert the normalized array into a dataframe
normalized_df = pd.DataFrame(normalized_df, columns
=df.columns)
#print the normalized dataframe
print("Normalized dataframe:")
print(normalized_df)
```

The original data frame, is as follows:

```
   age  height  weight
0   25     160      60
1   35     170      70
2   45     180      80
3   55     190      90
```

The normalized data frame, is as follows:

```
        age    height    weight
0  0.000000  0.000000  0.000000
1  0.333333  0.333333  0.333333
2  0.666667  0.666667  0.666667
3  1.000000  1.000000  1.000000
```

Tutorial 2.14. An example to read a Comma Separated File (CSV) and normalize the selected column in it using **pandas**

and **sklearn** library is as follows, using the **diabetes.csv** data:

```
# import MinMaxScaler class from the preprocessing module of scikit-learn library for data normalization
from sklearn.preprocessing import MinMaxScaler
import pandas as pd
# import IPython.display for displaying the dataframe
from IPython.display import display
# read the csv file from the directory and store it as a dataframe
diabetes_df = pd.read_csv(
    "/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
# specify the columns to normalize, which are all the numerical features in the dataframe
columns_to_normalize = ['Pregnancies', 'Glucose', 'BloodPressure',
                        'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome']
# display the unnormalized dataframe
display(diabetes_df[columns_to_normalize].head(4))
# create an instance of MinMaxScaler object that can normalize the data
scaler = MinMaxScaler()
# fit and transform the dataframe using the scaler and assign the normalized values to the same columns
diabetes_df[columns_to_normalize] = scaler.fit_transform(
    diabetes_df[columns_to_normalize])
# print a message to indicate the normalized structured data
print("Normalized Structured Data:")
```

```
# display the normalized dataframe
display(diabetes_df.head(4))
```

The output of *Tutorial 2.14* will be a data frame with normalized values in the selected columns.

Data standardization

Data standardization is a type of data transformation that adjusts data to have a mean of zero and a standard deviation of one. It helps compare variables with different scales or units and is necessary for algorithms like **Principal Component Analysis (PCA)**, **Linear Discriminant Analysis (LDA)**, or k-means clustering that require standardized data. By standardizing values, we can measure how far each value is from the mean in terms of standard deviations. This can help us identify outliers, perform hypothesis tests, or apply machine learning algorithms that require standardized data. There are different ways to standardize data like min-max normalization described in normalization of data frames, but the z-score formula remains the most widely used. This formula adjusts each value in a dataset by subtracting the mean and dividing it by the standard deviation. The formula is as follows:

$$z = (x - \mu) / \sigma$$

Where x represents the original value, μ represents the mean, and σ represents the standard deviation.

Suppose, we have a dataset of two variables: height (in centimeters) and weight (in kilograms) of five people:

Height	Weight
160	50
175	70
180	80

168	60

Table 2.3: *Height and weight of peoples*

The mean height is 169 cm and the standard deviation is 7.6 cm. The mean weight is 62.4 kg and the standard deviation is 11.6 kg. To standardize the data, we use the formula as follows:

$$z = (x - \mu) / \sigma$$

where x is the original value, μ is the mean, and σ is the standard deviation. Applying this formula to each value in the dataset, we get the following standardized values:

Height (z-score)	Weight (z-score)
-1.18	-1.07
0.79	0.66
1.45	1.52
-0.13	-0.21

Table 2.4: *Standardized height and weight*

Now, the two variables have an average of zero and a standard deviation of one, and they are measured on the same scale. The standardized values reflect the extent to which each observation deviates from the mean in terms of standard deviations.

Standardization of NumPy array

Tutorial 2.15. An example to show standardization of height and weight as a NumPy array, is as follows:

```
# Import numpy library for numerical calculations
import numpy as np
# Define the data as numpy arrays
height = np.array([160, 175, 180, 168, 162])
weight = np.array([50, 70, 80, 60, 52])
```

```
# Calculate the mean and standard deviation of each variable
height_mean = np.mean(height)
height_std = np.std(height)
weight_mean = np.mean(weight)
weight_std = np.std(weight)
# Define the z-score formula as a function
def z_score(x, mean, std):
    return (x - mean) / std
# Apply the z-score formula to each value in the data
height_z = z_score(height, height_mean, height_std)
weight_z = z_score(weight, weight_mean, weight_std)
# Print the standardized values
print("Height (z-score):", height_z)
print("Weight (z-score):", weight_z)
```

Output:

```
Height (z-score): [-1.18421053  0.78947368  1.44736842 -0.13157895 -0.92105263]
Weight (z-score): [-1.06904497  0.65465367  1.51887505 -0.20655562 -0.89792798]
```

Standardization of data frame

Tutorial 2.16. An example to show standardization of height and weight as a data frame, is as follows:

```
# Import pandas library for data manipulation
import pandas as pd
# Define the original data as a pandas dataframe
data = pd.DataFrame({"Height": [160, 175, 180, 168, 162],
"Weight": [50, 70, 80, 60, 52]})
```

```
# Calculate the mean and standard deviation of each column
data_mean = data.mean()
data_std = data.std()
# # Define the z-score formula as a function
def z_score(column):
    mean = column.mean()
    std_dev = column.std()
    standardized_column = (column - mean) / std_dev
    return standardized_column
# Apply the z-score formula to each column in the dataframe
data_z = data.apply(z_score)
# Print the standardized dataframe
print("Data (z-score):", data_z)
```

Output:

```
Data (z-score):      Height    Weight
0 -1.060660 -0.984003
1  0.707107  0.603099
2  1.296362  1.396649
3 -0.117851 -0.190452
4 -0.824958 -0.825293
```

Data transformation

Data transformation is essential as it satisfies the requirements for particular statistical tests, enhances data interpretation, and improves the visual representation of charts. For example, consider a dataset that includes the heights of 100 students measured in centimeters. If the distribution of data is positively skewed (more students are shorter than taller), assumptions like normality and equal variances must be satisfied before conducting a t-test. A t-

test (a statistical test used to compare the means of two groups) on the average height of male and female students may produce inaccurate results if skewness violates these assumptions.

To mitigate this problem, transform the height data by taking the square root or logarithm of each measurement. Doing so will improve consistency and accuracy. Perform a t-test on the transformed data to compute the average height difference between male and female students with greater accuracy. Use the inverse function to revert the transformed data back to its original scale. For example, if the transformation involved the square root, then square the result to express centimeters. Another reason to use data transformation is to improve data visualization and understanding. For example, suppose you have a dataset of the annual income of 1000 people in US dollars that is skewed to the right, indicating that more participants are in the lower-income bracket. If you want to create a histogram that shows income distribution, you will see that most of the data is concentrated in a few bins on the left, while some outliers exist on the right side. For improved clarity in identifying the distribution pattern and range, apply a transformation to the income data by taking the logarithm of each value. This distributes the data evenly across bins and minimizes the effect of outliers. After that, plot a histogram of the log-transformed income to show the income fluctuations among individuals.

Tutorial 2.17: An example to show the data transformation of the annual income of 1000 people in US dollars, which is a skewed data set, is as follows:

```
# Import the libraries
import numpy as np
import matplotlib.pyplot as plt
# Generate some random data for the annual income of 1000 people in US dollars
```

```
np.random.seed(42) # Set the seed for reproducibility
income = np.random.lognormal(mean=10, sigma=1, size=1000) # Generate 1000 incomes from a lognormal distribution with mean 10 and standard deviation 1
income = income.round(2) # Round the incomes to two decimal places
# Plot a histogram of the original income
plt.hist(income, bins=20)
plt.xlabel("Income (USD)")
plt.ylabel("Frequency")
plt.title("Histogram of Income")
plt.show()
```

Suppose the initial actual distribution of annual income of 1000 people in US dollars as shown in *Figure 2.1:*

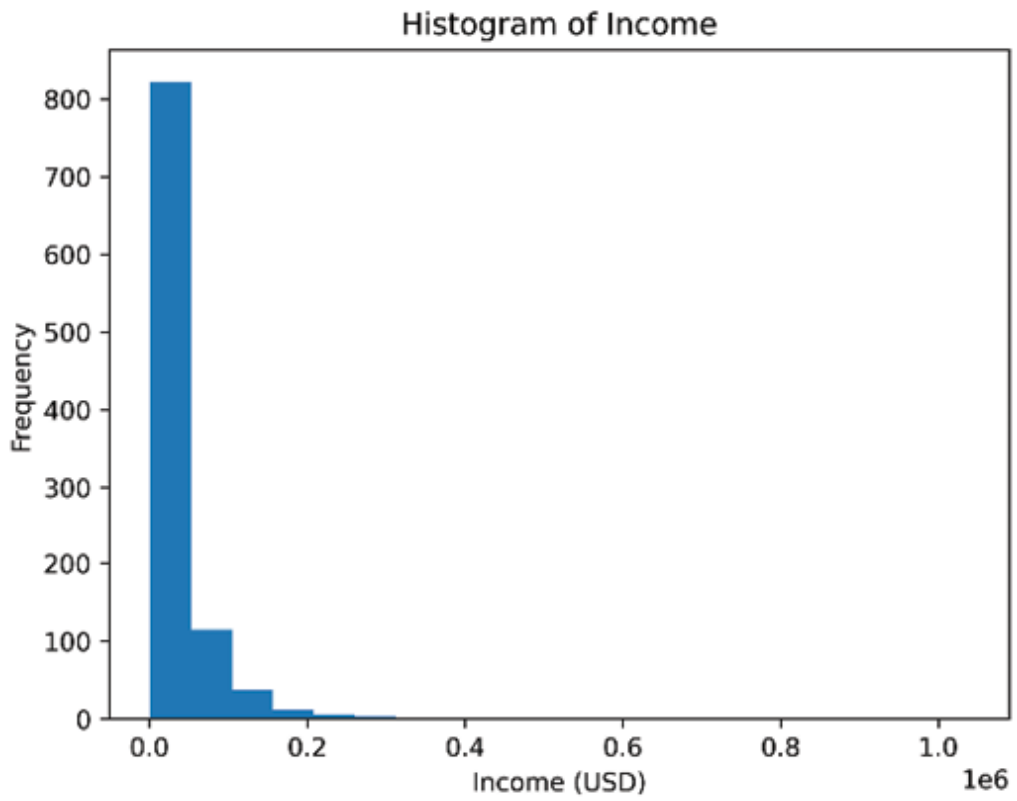

Figure 2.1: *Distribution of annual income of 1000 people in US dollars*

Now, let us apply the logarithmic transformation to the income:

```
# Apply a logarithm transformation to the income
log_income = np.log10(income) # Take the base 10 logarithm of each income value
# Plot a histogram of the transformed income
plt.hist(log_income, bins=20)
plt.xlabel("Logarithm of Income")
plt.ylabel("Frequency")
plt.title("Histogram of Logarithm of Income")
# Set the DPI to 600
plt.savefig('data_transformation2.png', dpi=600)
# Show the plot (optional)
plt.show()
```

The **log10()** function in the above code takes the base 10 logarithm of each income value. This means that it converts the income values from a linear scale to a logarithmic scale, where each unit increase on the x-axis corresponds to a 10-fold increase on the original scale. For example, if the income value is 100, the **log10** value is 2, and if the income value is 1000, the **log10** value is 3.

The **log10** function is useful for data transformation because it can reduce the skewness and variability of the data, and make it easier to compare values that differ by orders of magnitude.

Now, let us plot the histogram of income after logarithmic transformation as follows:

```
# Label the x-
axis with the original values by using 10^x as tick marks
plt.hist(log_income, bins=20)
plt.xlabel("Income (USD)")
plt.ylabel("Frequency")
```

```
5. plt.title("Histogram of Logarithm of Income")
6. plt.xticks(np.arange(1, 7), ["$10", "$100", "$1K", "$10K",
   "$100K", "$1M"])
7. plt.show()
```

The histogram of logarithm of income with original values is plotted as shown in *Figure 2.2*:

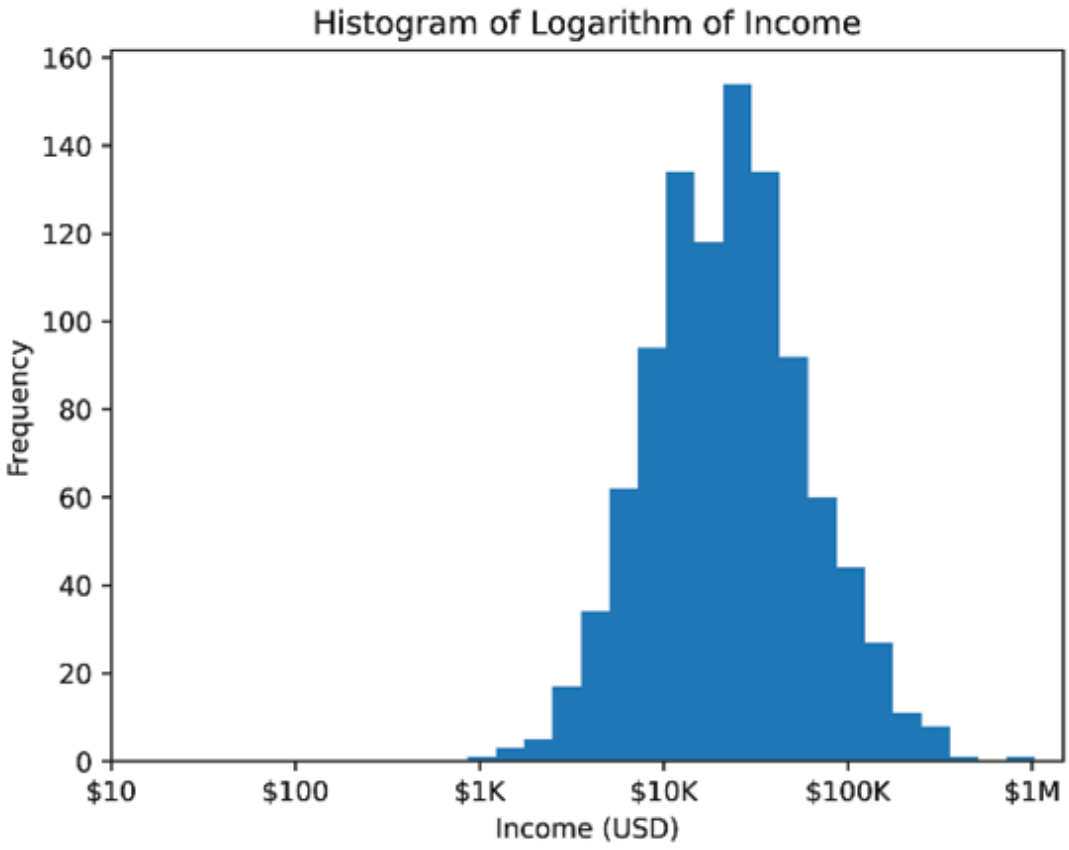

Figure 2.2: *Logarithmic distribution of annual income of 1000 people in US dollars*

As you can see, the data transformation made the data more evenly distributed across bins, and reduced the effect of outliers. The histogram of the log-transformed income showed a clearer picture of how income varies among people.

In unstructured data like text, normalization may involve natural language processing like convert lowercase,

removing punctuation, handling special character like whitespaces and many more. In image or audio, it may involve rescaling pixel values, extracting features.

Tutorial 2.18: An example to convert lowercase, removing punctuation, handling special character like whitespaces in unstructured text data, is as follows:

```
# Import the re module, which provides regular expression operations
import re

# Define a function named normalize_text that takes a text as an argument
def normalize_text(text):
    # Convert all the characters in the text to lowercase
    text = text.lower()
    # Remove any punctuation marks (such as . , ! ?) from the text using a regular expression
    text = re.sub(r'[^\w\s]', '', text)
    # Remove any extra whitespace (such as tabs, newlines, or multiple spaces) from the text using a regular expression
    text = re.sub(r'\s+', ' ', text).strip()
    # Return the normalized text as the output of the function
    return text

# Create a sample unstructured text data as a string
unstructured_text = "This is an a text for book Implementing Stat with Python, with! various punctuation marks... "

# Call the normalize_text function on the unstructured text and assign the result to a variable named normalized
```

```
_text
normalized_text = normalize_text(unstructured_text)
# Print the original and normalized texts to compare the
m
print("Original Text:", unstructured_text)
print("Normalized Text:", normalized_text)
```

Output:

```
Original Text: This is an a text for book Implementing
Stat with Python, with! various punctuation marks...
Normalized Text: this is an a text for book implementing
stat with python with various punctuation marks
```

Data binning, grouping, encoding

Data binning, grouping, and encoding are common data preprocessing and feature engineering techniques. They transform the original data into a format suitable for modeling or analysis.

Data binning

Data binning groups continuous or discrete values into a smaller number of bins or intervals. For example, if you have data on the ages of 100 people, you may group them into five bins: [0-20), [20-40), [40-60), [60-80), and [80-100], where [0-20) includes values greater than or equal to 0 and less than 20, [80-100] includes values greater than or equal to 80 and less than or equal to 100. Each bin represents a range of values, and the number of cases in each bin can be counted or visualized. Data binning reduces noise, outliers, and skewness in the data, making it easier to view distribution and trends.

Tutorial 2.19: A simple implementation of data binning for grouping the ages of 100 people into five bins: [0-20), [20-40), [40-60), [60-80), and [80-100] is as follows:

```
# Import the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Generate some random data for the ages of 100 people
np.random.seed(42) # Set the seed for reproducibility
ages = np.random.randint(low=0, high=101, size=100) # Generate 100 ages between 0 and 100
# Create a pandas dataframe with the ages
df = pd.DataFrame({"Age": ages}) # Create a dataframe with one column: Age
# Define the bins and labels for the age groups
bins = [0, 20, 40, 60, 80, 100] # Define the bin edges
labels = ["[0-20)", "[20-40)", "[40-60)", "[60-80)", "[80-100]"] # Define the bin labels
# Apply data binning to the ages using the pd.cut function
df["Age Group"] = pd.cut(df["Age"], bins=bins, labels=labels, right=False) # Create a new column with the age groups
# Print the first 10 rows of the dataframe
print(df.head(10))
```

Output:

```
   Age Age Group
0   51   [40-60)
1   92  [80-100]
2   14    [0-20)
3   71   [60-80)
4   60   [60-80)
5   20   [20-40)
6   82  [80-100]
```

```
9. 7   86  [80-100]
10. 8   74   [60-80)
11. 9   74   [60-80)
```

Tutorial 2.20: An example to apply binning on diabetes dataset by grouping the ages of all the people in dataset into three bins: [< 30], [30-60], [60-100], is as follows:

```
1. import pandas as pd
2. # Read the json file from the direcotory
3. diabetes_df = pd.read_csv(
4.     "/workspaces/ImplementingStatisticsWithPython/data
   /chapter1/diabetes.csv")
5. # Define the bin intervals
6. bin_edges = [0, 30, 60, 100]
7. # Use cut to create a new column with bin labels
8. diabetes_df['Age_Group'] = pd.cut(diabetes_df['Age'],
    bins=bin_edges, labels=[
9.                              '<30', '30-60', '60-100'])
10. # Count the number of people in each age group
11. age_group_counts = diabetes_df['Age_Group'].
   value_counts().sort_index()
12. # View new DataFrame with the new bin(categories) colu
   mns
13. diabetes_df
```

The output is a new data frame with **Age_Group** column consisting appropriate bin label.

Tutorial 2.21: An example to apply binning on NumPy array data by grouping the scores of students in exam into five bins based on the scores obtained: [< 60], [60-69], [70-79], [80-89] , [90+], is as follows:

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # Create a sample NumPy array of exam scores
```

```
scores = np.array([75, 82, 95, 68, 90, 85, 78, 72, 88, 93, 60, 72, 80])
# Define the bin intervals
bin_edges = [0, 60, 70, 80, 90, 100]
# Use histogram to count the number of scores in each bin
bin_counts, _ = np.histogram(scores, bins=bin_edges)
# Plot a histogram of the binned scores
plt.bar(range(len(bin_counts)), bin_counts, align='center')
plt.xticks(range(len(bin_edges) - 1), ['<60', '60-69', '70-79', '80-89', '90+'])
plt.xlabel('Score Range')
plt.ylabel('Number of Scores')
plt.title('Distribution of Exam Scores')
plt.savefig("data_binning2.jpg",dpi=600)
plt.show()
```

Output:

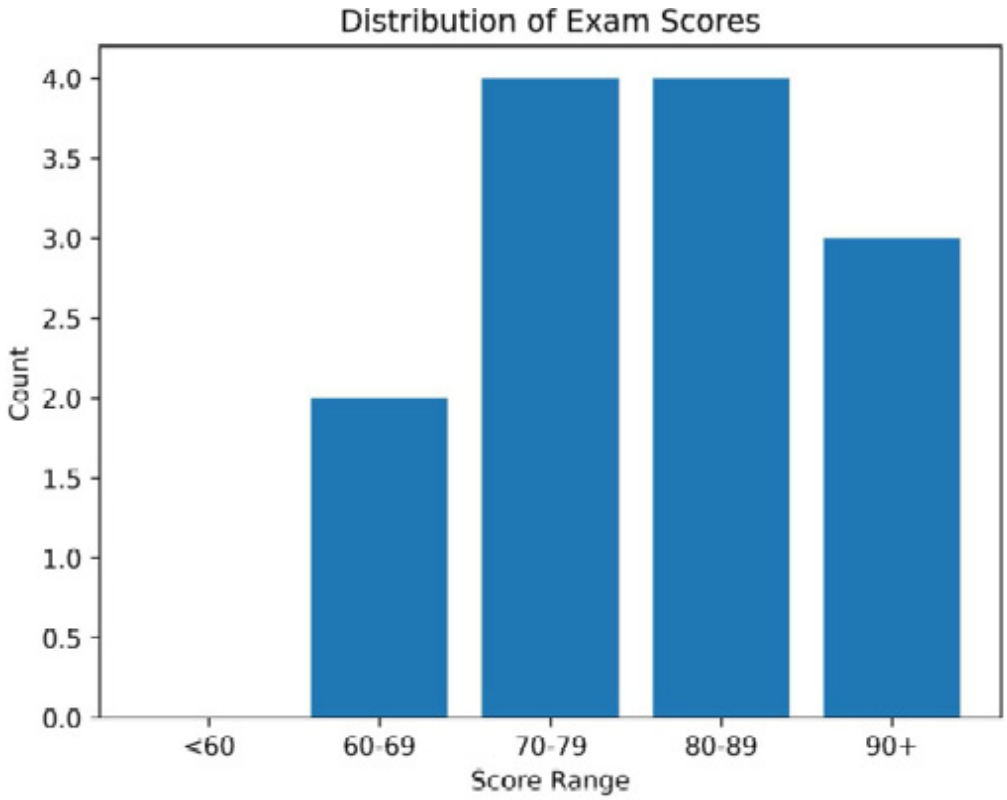

Figure 2.3: *Distribution of student's exam scores in five bins*

In text files, data binning can be grouping and categorizing of text data based on some criteria. To apply data binning on the text data, keep the following points in mind:

- Determine a criterion for binning. For example: Could be count of sentences in text, word count, sentiment score, topic.
- Read text and calculate the selected criteria for binning. For example: Count number of words in bins.
- Define bins based on range of values for the selected criteria. For example: Defining short, medium, long based on word count of text.
- Assign text files appropriate bin based on calculated value.
- Analyze or summarize the data in the new bins.

Some use cases of binning in text file are grouping text files based on their length, binning based on the sentiment

analysis score, topic binning by performing topic modelling, language binning if text files are in different languages, time-based binning if text files have timestamps.

Tutorial 2.22: An example showing data binning of text files using word counts in the files with three bins: [<26 words] as short [26 and 30 words (inclusive)] as medium, [>30] as long, is as follows:

```
# Import the os, glob, and pandas modules
import os
import glob
import pandas as pd
# Define the path of the folder that contains the files
path = "/workspaces/ImplementingStatisticsWithPython/data/chapter1/TransactionNarrative"
files = glob.glob(path + "/*.txt") # Get a list of files that match the pattern "/*.txt" in the folder
# Display a the information in first file
file_one = glob.glob("/workspaces/ImplementingStatisticsWithPython/data/chapter1/TransactionNarrative/1.txt")
for file1 in file_one: # Loop through the file_one list
    # To open each file in read mode with utf-8 encoding and assign it to a file object
    with open(file1, "r", encoding="utf-8") as f1: # Open each file in read mode with utf-8 encoding and assign it to a file object named f1
        print(f1.read()) # Print the content of the file object
# Function that takes a file name as an argument and returns the word count of that file
def word_count(file): # Define a function named word_count that takes a file name as an argument
    # Open the file in read mode
```

```
    with open(file, "r") as f: # Open the file in read mode and assign it to a file object named f
        # Read the file content
        content = f.read() # Read the content of the file object and assign it to a variable named content
        # Split the content by whitespace characters
        words = content.split() # Split the content by white space characters and assign it to a variable named words
        # Return the length of the words list
        return len(words) # Return the length of the words list as the output of the function
counts = [word_count(file) for file in files] # Use a list comprehension to apply the word_count function to each file in the files list and assign it to a variable named counts
binning_df = pd.DataFrame({"file": files, "count": counts}) # Create a pandas dataframe with two columns: file and count, using the files and counts lists as values
binning_df["bin"] = pd.cut(binning_df["count"], bins=[0, 26, 30, 35]) # Create a new column named bin, using the pd.cut function to group the count values into three bins: [0-26), [26-30), and [30-35]
binning_df["bin"] = pd.cut(binning_df["count"], bins=[0, 26, 30, 35], labels=["Short", "Medium", "Long"]) # Replace the bin values with labels: Short, Medium, and Long, using the labels argument of the pd.cut function
binning_df # Display the dataframe
```

Output:

The output shows a sample text file, then, the file names, the number of words in each file, and the assigned bin labels as follows:

```
Date: 2023-08-05
```

2. Merchant: Bistro Delight
3. Amount: $42.75
4. Description: Dinner with colleagues - celebrating a successful project launch.
5.
6. Thank you for choosing Bistro Delight. Your payment of $42.75 has been processed.
7.
8. file count bin
9. 0 /workspaces/ImplementingStatisticsWithPython/d... 25 Short
10. 1 /workspaces/ImplementingStatisticsWithPython/d... 30 Medium
11. 2 /workspaces/ImplementingStatisticsWithPython/d... 31 Long
12. 3 /workspaces/ImplementingStatisticsWithPython/d... 27 Medium
13. 4 /workspaces/ImplementingStatisticsWithPython/d... 33 Long

In unstructured data, the data binning can be used for text categorization and modelling of text data, color quantization and feature extraction on image data, audio segmentation and feature extraction on audio data.

Data grouping

Data grouping aggregates data by criteria or categories. For example, if sales data exists for different products or market regions, grouping by product type or region can be beneficial. Each group represents a subset of data that shares some common attribute, allowing for comparison of summary statistics or measures. Data grouping simplifies information, emphasizes group differences or similarities, and exposes patterns or relationships.

Tutorial 2.23: An example for grouping sales data by product and region for three different products, is as follows:

```
# Import pandas library
import pandas as pd
# Create a sample sales data frame with columns for product, region, and sales
sales_data = pd.DataFrame({
    "product": ["A", "A", "B", "B", "C", "C"],
    "region": ["North", "South", "North", "South", "North", "South"],
    "sales": [100, 200, 150, 250, 120, 300]
})
# Print the sales data frame
print("\nOriginal dataframe")
print(sales_data)
# Group the sales data by product and calculate the total sales for each product
group_by_product = sales_data.groupby("product").sum()
# Print the grouped data by product
print("\nGrouped by product")
print(group_by_product)
# Group the sales data by region and calculate the average sales for each region
group_by_region = sales_data.groupby("region").sum()
# Print the grouped data by region
print("\nGrouped by region")
print(group_by_region)
```

Output:

```
Original dataframe
```

```
  product region  sales
0       A  North    100
1       A  South    200
2       B  North    150
3       B  South    250
4       C  North    120
5       C  South    300

Grouped by product
            region  sales
product
A       NorthSouth    300
B       NorthSouth    400
C       NorthSouth    420

Grouped by region
       product  sales
region
North      ABC    370
South      ABC    750
```

Tutorial 2.24: An example to show grouping of data based on age interval through binning and calculate the mean score for each group, is as follows:

```
# Import pandas library to work with data frames
import pandas as pd
# Create a data frame with student data, including name, age, and score
data = {'Name': ['John', 'Anna', 'Peter', 'Carol', 'David', 'Oystein','Hari'],
        'Age': [15, 16, 17, 15, 16, 14, 16],
        'Score': [85, 92, 78, 80, 88, 77, 89]}
df = pd.DataFrame(data)
```

```
# Create age intervals based on the age column, using bins of 13-16 and 17-18
age_intervals = pd.cut(df['Age'], bins=[13, 16, 18])
# Group the data frame by the age intervals and calculate the mean score for each group
grouped_data = df.groupby(age_intervals)['Score'].mean()
# Print the grouped data with the age intervals and the mean score
print(grouped_data)
```

Output:

```
Age
(13, 16]    85.166667
(16, 18]    78.000000
Name: Score, dtype: float64
```

Tutorial 2.25: An example of grouping a scikit-learn digit image dataset based on target labels, where target labels are numbers from 0 to 9, is as follows:

```
# Import the sklearn library to load the digits dataset
from sklearn.datasets import load_digits
# Import the matplotlib library to plot the images
import matplotlib.pyplot as plt

# Class to display and perform grouping of digits
class Digits_Grouping:
    # Contructor method to initialize the object's attributes
    def __init__(self, digits):
        self.digits = digits

    def display_digit_image(self):
```

```
        # Get the images and labels from the dataset
        images = self.digits.images
        labels = self.digits.target
        # Display the first few images along with their label
s
        num_images_to_display = 5  # You can change this
number as needed
        # Plot the selected few image in a subplot
        plt.figure(figsize=(10, 4))
        for i in range(num_images_to_display):
            plt.subplot(1, num_images_to_display, i + 1)
            plt.imshow(images[i], cmap='gray')
            plt.title(f"Label: {labels[i]}")
            plt.axis('off')
        # Save the figure to a file with no padding
        plt.savefig('data_grouping.jpg', dpi=600, bbox_inch
es='tight')
        plt.show()

    def display_label_based_grouping(self):
        # Group the data based on target labels
        grouped_data = {}
        # Iterate through each image and its corresponding
target in the dataset.
        for image, target in zip(self.digits.images, self.digit
s.target):
            # Check if the current target value is not already
present as a key in grouped_data.
            if target not in grouped_data:
                # If the target is not in grouped_data, add it as
a new key with an empty list as the value.
```

```
            grouped_data[target] = []
        # Append the current image to the list associated with the target key in grouped_data.
        grouped_data[target].append(image)
    # Print the number of samples in each group
    for target, images in grouped_data.items():
        print(f"Target {target}: {len(images)} samples")

# Create an object of Digits_Grouping class with the digits dataset as an argument
displayDigit = Digits_Grouping(load_digits())
# Call the display_digit_image method to show some images and labels from the dataset
displayDigit.display_digit_image()
# Call the display_label_based_grouping method to show how many samples are there for each label
displayDigit.display_label_based_grouping()
```

Output:

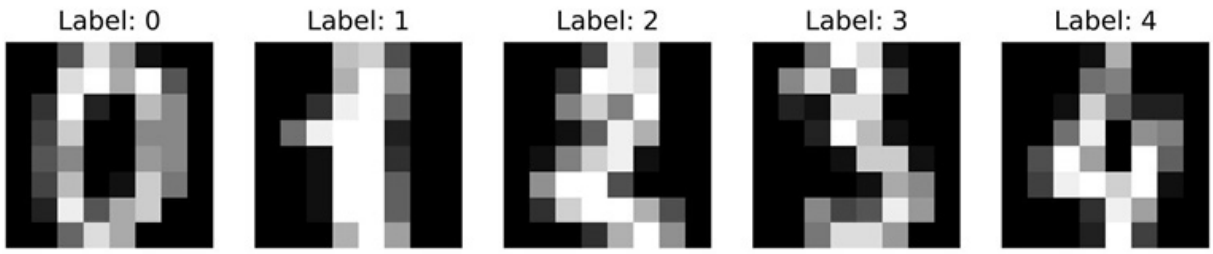

Figure 2.4: *Images and respective labels of digit dataset*

```
Target 0: 178 samples
Target 1: 182 samples
Target 2: 177 samples
Target 3: 183 samples
Target 4: 181 samples
Target 5: 182 samples
```

```
Target 6: 181 samples
Target 7: 179 samples
Target 8: 174 samples
Target 9: 180 samples
```

Data encoding

Data encoding converts categorical or text-based data into numeric or binary form. For example, you can encode gender data of 100 customers as 0 for male and 1 for female. This encoding corresponds to a specific value or level of the categorical variable to assist machine learning algorithms and statistical models. Encoding data helps manage non-numeric data, reduces data dimensionality, and enhances model performance. It is useful because it allows us to convert data from one form to another, usually for the purpose of transmission, storage, or analysis. Data encoding can help us prepare data for analysis, develop features, compress data, and protect data.

There are several techniques for encoding data, depending on the type and purpose of the data as follows:

- **One-hot encoding:** This technique converts categorical variables, which have a finite number of discrete values or categories, into binary vectors of 0s and 1s. Each category is represented by a unique vector where only one element is 1 and the rest are 0. Appropriate when ordinality is important. One-hot encoding generates a column for every unique category variable value, and binary 1 or 0 values indicate the presence or absence of each value in each row. This approach encodes categorical data in a manner that facilitates comprehension and interpretation by machine learning algorithms. Nevertheless, it expands data dimensions and produces sparse matrices.

Tutorial 2.26: An example of applying one-hot encoding in gender and color, is as follows:

```
import pandas as pd
# Create a sample dataframe with 3 columns: name, gender and color
df = pd.DataFrame({
    'name': ['Alice', 'Eve', 'Lee', 'Dam', 'Eva'],
    'gender': ['F', 'F', 'M', 'M', 'F'],
    'color': ['yellow', 'green', 'green', 'yellow', 'pink']
})
# Print the original dataframe
print("Original dataframe")
print(df)
# Apply one hot encoding on the gender and color columns using pandas.get_dummies()
df_encoded = pd.get_dummies(df, columns=['gender', 'color'], dtype=int)
# Print the encoded dataframe
print("One hot encoded dataframe")
df_encoded
```

Tutorial 2.27: An example of applying one-hot encoding in object data type column in data frame using UCI adult dataset, is as follows:

```
import pandas as pd
import numpy as np
# Read the json file from the direcotory
diabetes_df = pd.read_csv(
    "/workspaces/ImplementingStatisticsWithPython/data/chapter2/Adult_UCI/adult.data")

# Define a function for one hot encoding
def one_hot_encoding(diabetes_df):
    # Identify columns that are categorical to apply one h
```

```
    ot encoding in them only
    columns_for_one_hot = diabetes_df.select_dtypes(include="object").columns
    # Apply one hot encoding to the categorical columns
    diabetes_df = pd.get_dummies(
        diabetes_df, columns=columns_for_one_hot, prefix=columns_for_one_hot, dtype=int)
    # Display the transformed dataframe
    print(display(diabetes_df.head(5)))

# Call the one hot encoding method by passing dataframe as argument
one_hot_encoding(diabetes_df)
```

- **Label coding:** This technique assigns a numeric value to each category of a categorical variable. The numerical values are usually sequential integers starting from 0. Appropriate when order is important. The transformed variable will have numerical values instead of categorical values. Its drawback is the loss of information about the similarity or difference between categories.

Tutorial 2.28: An example of applying label encoding for categorical variables, is as follows:

```
import pandas as pd
# Create a data frame with name, gender, and color columns
df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Ane', 'Bo'],
    'gender': ['F', 'M', 'M', 'M', 'F', 'F', 'M'],
    'color': ['red', 'blue', 'green', 'yellow', 'pink', 'red', 'blue']
})
```

```
# Convert the gender column to a categorical variable and assign numerical codes to each category
df['gender_label'] = df['gender'].astype('category').cat.codes
# Convert the color column to a categorical variable and assign numerical codes to each category
df['color_label'] = df['color'].astype('category').cat.codes
# Print the data frame with the label encoded columns
print(df)
```

- **Binary encoding:** Binary coding converts categorical variables into fixed-length binary codes. Performing a binary search on sorted categories records the comparison result as 1 or 0. Each unique category is assigned an integer value, which is then converted into binary code. This reduces the number of columns necessary to describe categorical data, unlike one-hot encoding, which requires a new column for each unique category. However, binary encoding has certain downsides, such as the creation of ordinality or hierarchy within categories that did not previously exist, making interpretation and analysis more challenging.

Tutorial 2.29: An example of applying binary encoding for categorical variables using **category_encoders** package from **pip**, is as follows:

```
# Import pandas library and category_encoders library
import pandas as pd
import category_encoders as ce
# Create a sample dataframe with 3 columns: name, gender and color
df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Ane', 'Bo'],
    'gender': ['F', 'M', 'M', 'M', 'F', 'F', 'M'],
```

```
    'color': ['red', 'blue', 'green', 'yellow', 'pink', 'red', 'blue']
})
# Print the original dataframe
print("Original dataframe")
print(df)
# Create a binary encoder object
encoder = ce.BinaryEncoder(cols=['gender', 'color'])
# Fit and transform the dataframe using the encoder
df_encoded = encoder.fit_transform(df)
# Print the encoded dataframe
print("Binary encoded dataframe")
print(df_encoded)
```

Output:

```
Original dataframe
      name gender   color
0    Alice      F     red
1      Bob      M    blue
2  Charlie      M   green
3    David      M  yellow
4      Eve      F    pink
5      Ane      F     red
6       Bo      M    blue
Binary encoded dataframe
      name  gender_0  gender_1  color_0  color_1  color_2
0    Alice         0         1        0        0        1
1      Bob         1         0        0        1        0
2  Charlie         1         0        0        1        1
3    David         1         0        1        0        0
4      Eve         0         1        1        0        1
5      Ane         0         1        0        0        1
```

18. 6 Bo 1 0 0 1 0

The difference between binary encoders and one-hot encoders is in how they encode categorical variables. One-hot encoding, which creates a new column for each categorical value and marks their existence with either 1 or 0. However, binary encoding converts each categorical variable value into a binary code and separates them into distinct columns. For example, data frame's color column can be one-hot encoded, as shown below.

The same color column of the data frame as can be binary encoded, where each unique combination of bits represents a specific color, as follows:

Tutorial 2.30: An example to illustrate difference of one-hot encoding and binary encoding, is as follows:

```
# Import the display function to show the data frames
from IPython.display import display
# Import pandas library to work with data frames
import pandas as pd
# Import category_encoders library to apply different encoding techniques
import category_encoders as ce

# Class to compare the difference between one-hot encoding and binary encoding
class Encoders_Difference:
    # Constructor method to initialize the object's attribute
    def __init__(self, df):
        self.df = df

    # Method to apply one-hot encoding to the color column
```

```
    def one_hot_encoding(self):
        # Use the get_dummies function to create binary vectors for each color category
        df_encoded1 = pd.get_dummies(df, columns=['color'], dtype=int)
        # Display the encoded data frame
        print("One-hot encoded dataframe")
        print(df_encoded1)

    # Method to apply binary encoding to the color column
    def binary_encoder(self):
        # Create a binary encoder object with the color column as the target
        encoder = ce.BinaryEncoder(cols=['color'])
        # Fit and transform the data frame with the encoder object
        df_encoded2 = encoder.fit_transform(df)
        # Display the encoded data frame
        print("Binary encoded dataframe")
        print(df_encoded2)

# Create a sample data frame with 3 columns: name, gender and color
df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Ane'],
    'gender': ['F', 'M', 'M', 'M', 'F', 'F'],
    'color': ['red', 'blue', 'green', 'blue', 'green', 'red']
})

# Create an object of Encoders_Difference class with the sample data frame as an argument
```

```
encoderDifference_obj = Encoders_Difference(df)
# Call the one_hot_encoding method to show the result of one-hot encoding
encoderDifference_obj.one_hot_encoding()
# Call the binary_encoder method to show the result of binary encoding
encoderDifference_obj.binary_encoder()
```

Output:

```
One-hot encoded dataframe
      name gender  color_blue  color_green  color_red
0    Alice      F           0            0          1
1      Bob      M           1            0          0
2  Charlie      M           0            1          0
3    David      M           1            0          0
4      Eve      F           0            1          0
5      Ane      F           0            0          1
Binary encoded dataframe
      name gender  color_0  color_1
0    Alice      F        0        1
1      Bob      M        1        0
2  Charlie      M        1        1
3    David      M        1        0
4      Eve      F        1        1
5      Ane      F        0        1
```

- **Hash coding:** This technique applies a hash function to each category of a categorical variable and maps it to a numeric value within a predefined range. The hash function is typically a one-way function that produces a unique output for each input.
- **Feature scaling:** This technique transforms numerical variables into a common scale or range, usually between 0 and 1 or -1 and 1. Different methods of feature scaling,

such as min-max scaling, standardization, and normalization, are discussed above.

Missing data, detecting and treating outliers

Data values that are not stored or captured for some variables or observations in a dataset are referred to as missing data. It may happen for a number of reasons, including human mistakes, equipment malfunctions, data entry challenges, privacy concerns, or flaws with survey design. The accuracy and reliability of the analysis and inference can be impacted by missing data. In structured data, identifying missing values is pretty easy whereas in semi and unstructured it may not always be the case.

Tutorial 2.31: An example to illustrate how to count sum of all the null and missing values in large data frame, is as follows:

```
import pandas as pd
# Create a dataframe with some null values
df = pd.DataFrame({"Name": ["Alice", "Bob", "Charlie", None, "Eve"],
                   "Age": [25, 30, 35, None, 40],
                   "Gender": ["F", "M", None, None, "F"]})
# Display the dataframe
print("Original dataframe")
print(df)
# Use isna().sum() to view the sum of null values for eac
h column
print("Null value count in dataframe")
print(df.isna().sum())
```

Output:

```
Original dataframe
      Name   Age Gender
```

```
0    Alice  25.0      F
1      Bob  30.0      M
2  Charlie  35.0   None
3     None   NaN   None
4      Eve  40.0      F
Null value count in dataframe
Name      1
Age       1
Gender    2
```

Some of the most common techniques to handle missing data are deletion of missing data row or column, imputation of missing value and prediction of missing value.

Tutorial 2.32: An example to show all columns in data frame and remaining columns after applying drop, is as follows:

```
import pandas as pd
# Read the json file from the direcotory
diabetes_df = pd.read_csv(
    "/workspaces/ImplementingStatisticsWithPython/data/chapter2/Adult_UCI/adult.data")
# View all columns in dataframe
print("Columns before drop")
print(diabetes_df.columns)
# Drop the 'Age' and 'Work' columns
diabetes_df = diabetes_df.drop(columns=
[' Work', ' person_id', ' education', ' education_number',
                                    ' marital_status'], axis=1)
# Verify the updated DataFrame
print("Columns after drop")
print(diabetes_df.columns)
```

Output:

```
Columns before drop
Index(['Age', ' Work', ' person_id', ' education', ' education_number',
       ' marital_status', ' occupation', ' relationship', ' race', ' gender',
       ' capital_gain', ' capital_loss', ' hours_per_week', ' native_country',
       ' income'],
      dtype='object')
Columns after drop
Index(['Age', ' occupation', ' relationship', ' race', ' gender',
       ' capital_gain', ' capital_loss', ' hours_per_week', ' native_country',
       ' income'],
      dtype='object')
```

Data imputation replaces missing or invalid data values with reasonable estimates, improving the quality and usability of data for analysis and modeling. For example, let us examine a data set that includes student grades in four subjects that is, Mathematics, English, Science, and History. However, some grades are either invalid or missing, as demonstrated in the following table:

Name	Math	English	Science	History
Ram	90	85	95	?
Deep	80	?	75	70
John	?	65	80	60
David	70	75	?	65

Table 2.5: *Grades of students in different subjects*

One easiest method for data imputation is by calculating the mean (average) of available values for each column. For

example, the mean of math is *(90 + 80 + 70) / 3 = 80*, the mean of English is *(85 + 65 + 75) / 3 = 75*, and so on. These means can be used to replace missing or invalid values with the corresponding mean values as shown in *Table 2.4*:

Name	Math	English	Science	History
Ram	90	85	95	**73.3**
Deep	80	**75**	75	70
John	**80**	65	80	60
David	70	75	**78.3**	65

Table 2.6: *Imputing missing scores based on mean*

Tutorial 2.33: An example to illustrate imputation of missing value in data frame with **mean()**, is as follows:

```
import pandas as pd
# Create a DataFrame with student data using a dictiona
ry
data = {'Name': ['John', 'Anna', 'Peter', 'Hari', 'Suresh', '
Ram'],
        'Age': [15, 16, np.nan, 16, 30, 31],
        'Score': [85, 92, 78, 80, np.nan, 76]}
student_DF = pd.DataFrame(data)
# Print a message before showing the dataframe with mi
ssing values
print(f'Before Mean Inputation DataFrame')
# Display the dataframe with missing values using the di
splay function
print(student_DF)
# Calculate the mean of the Age column and store it in a
 variable
mean_age = student_DF['Age'].mean()
# Calculate the mean of the Score column and store it in
```

```
 a variable
mean_score = student_DF['Score'].mean()
# Print a message before showing the dataframe with imputed values
print(f'DataFrame after mean imputation')
# Replace the missing values in the dataframe with the mean values using the fillna method and a dictionary
student_DF = student_DF.fillna(value={'Age': mean_age, 'Score': mean_score})
# Display the dataframe with imputed values using the display function
print(student_DF)
```

Output:

```
Before Mean Inputation DataFrame
     Name   Age  Score
0    John  15.0   85.0
1    Anna  16.0   92.0
2   Peter   NaN   78.0
3    Hari  16.0   80.0
4  Suresh  30.0    NaN
5     Ram  31.0   76.0
DataFrame after mean imputation
     Name   Age  Score
0    John  15.0   85.0
1    Anna  16.0   92.0
2   Peter  21.6   78.0
3    Hari  16.0   80.0
4  Suresh  30.0   82.2
5     Ram  31.0   76.0
```

In some cases, missing value prediction can be estimated and predicted based on other information available in the

data set. If the estimation is not done properly, it can introduce noise and uncertainty into the data. Missingness can also be used as a variable to indicate whether a value was missing or not. However, this can increase dimensionality. More about this is discussed in later chapters. Some general guidelines to handle missing values are as follows:

- If the missing data are randomly distributed in the data set and are not too many (less than 5% of the total observations), then a simple method such as replacing the missing values with the mean, median, or mode of the corresponding variable may be sufficient.
- If the missing data are not randomly distributed or are too many (more than 5% of the total observations), a simple method may introduce bias and reduce the variability of the data. In this case, a more sophisticated method that takes into account the relationship between variables may be preferable. For example, you can use a regression model to predict the missing values based on other variables, or a nearest neighbor approach to find the most similar observation and use its value as an imputation.
- If the missing data are longitudinal, that is, they occur in repeated measurements over time, then a method that takes into account the temporal structure of the data may be more appropriate. For example, one can use a time series model to predict the missing values based on past and future observations, or a mixed effects model to account for both fixed and random effects over time.

Visualization and plotting of data

Data visualization and plotting entail creating graphical representations of information, including charts, graphs, maps, and other visual aids. Using visual tools is imperative for comprehending intricate data and presenting

information captivatingly and efficiently. This is essential in recognizing patterns, trends, and anomalies in a dataset and conveying our discoveries effectively. For data visualization and plotting, there are various libraries available such as **Matplotlib**, **Seaborn**, **Plotly**, **Bokeh**, and **Vega-altair**, among others. When presenting information in a chart, the first step is to determine what type of chart is appropriate for the data. There are many factors to consider when choosing a chart type, such as the number of variables, the type of data the purpose of the analysis, and the preferences of the audience. To compare values within or between groups, utilize a bar graph, column graph, or bullet graph. These charts are effective for displaying distinctions, rankings, or proportions of categories. Pie charts, donut charts, and tree maps are effective for illustrating how data is composed of various components. These charts are useful for depicting percentages or fractions of a total.

A line, area or column chart is ideal for displaying temporal changes. These graphs are efficient in presenting trends, patterns, or fluctuations within a specific time frame. Use a scatter plot, bubble chart, or connected scatter plot to display the relationship between multiple variables. These charts effectively portray how variables are interconnected. To effectively display a data distribution across a range of values, consider utilizing a histogram, box plot, or scatter plot. These plots are ideal for illustrating the data's shape, spread, and outliers. The various types of plots are discussed as follows:

Line plot

Line plots are ideal for displaying trends and changes in continuous or ordered data points, especially for time series data that depicts how a variable evolves over time. For instance, one could use a line plot to monitor a patient's blood pressure readings taken at regular intervals

throughout the year, to monitor their health.

Tutorial 2.34: An example to plot patient blood pressure reading taken at different months of year using line plot, is as follows:

```
1. # Import matplotlib.pyplot module
2. import matplotlib.pyplot as plt
3. # Create a list of dates for the x-axis
4. dates = ["01/08/2023", "01/09/2023", "01/10/2023", "01/11/2023", "01/12/2023"]
5. # Create a list of blood pressure readings for the y-axis
6. bp_readings = [120, 155, 160, 170, 175]
7. # Plot the line plot with dates and bp_readings
8. plt.plot(dates, bp_readings)
9. # Add a title for the plot
10. plt.title("Patient's Blood Pressure Readings Throughout the Year")
11. # Add labels for the x-axis and y-axis
12. plt.xlabel("Date")
13. plt.ylabel("Blood Pressure (mmHg)")
14. # Show the plot
15. plt.savefig("lineplot.jpg", dpi=600, bbox_inches='tight')
16. plt.show()
```

Output:

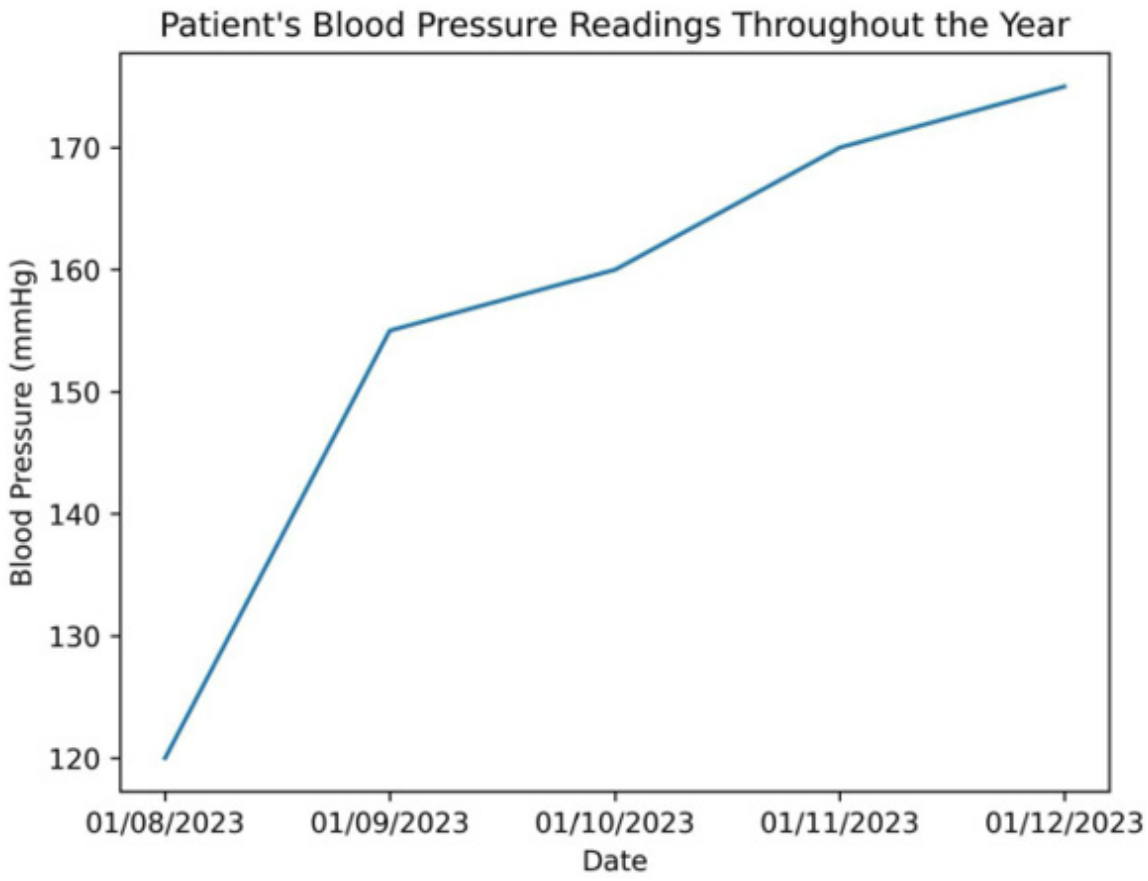

Figure 2.5: *Patient's blood pressure over the month in a line graph.*

Pie chart

Pie chart is useful when showing the parts of a whole and the relative proportions of different categories. Pie charts are best suited for categorical data with only a few different categories. Use pie charts to display the percentages of daily calories consumed from carbohydrates, fats, and proteins in a diet plan.

Tutorial 2.35: An example to display the percentages of daily calories consumed from carbohydrates, fats, and proteins in a pie chart, is as follows:

```
1.      # Import matplotlib.pyplot module
2. import matplotlib.pyplot as plt
3. # Create a list of dates for the x-axis
```

```
dates = ["01/08/2023", "01/09/2023", "01/10/2023", "01/11/2023", "01/12/2023"]
# Create a list of blood pressure readings for the y-axis
bp_readings = [120, 155, 160, 170, 175]
# Plot the line plot with dates and bp_readings
Plt.plot(dates, bp_readings)
# Add a title for the plot
plt.title("Patient's Blood Pressure Readings Throughout the Year")
# Add labels for the x-axis and y-axis
plt.xlabel("Date")
plt.ylabel("Blood Pressure (mmHg)")
# Show the plot
plt.savefig("lineplot.jpg", dpi=600, bbox_inches='tight')
plt.show()
```

Output:

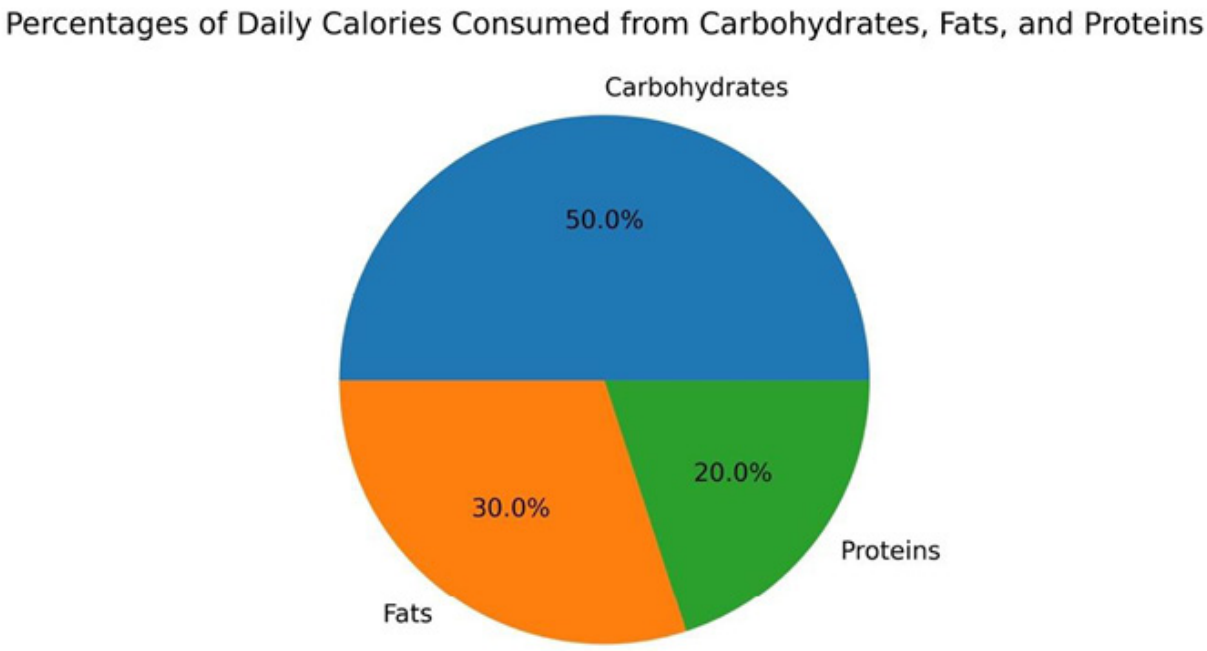

Figure 2.6: *Daily calories consumed from carbohydrates, fats, and proteins in a pie chart*

Bar chart

Bar charts are suitable for comparing values of different categories or showing the distribution of categorical data. Mostly useful for categorical data with distinct categories data type. For example: comparing the average daily step counts of people in their 20s, 30s, 40s, and so on, to assess the relationship between age and physical activity.

Tutorial 2.36: An example to plot average daily step counts of people in their 20s, 30s, 40s, and so on using bar chart, is as follows:

```
1. # Import matplotlib.pyplot module
2. import matplotlib.pyplot as plt
3.# Create a list of percentages of daily calories consumed
  from carbohydrates, fats, and proteins
4. calories = [50, 30, 20]
5. # Create a list of labels for the pie chart
```

```
6. labels = ["Carbohydrates", "Fats", "Proteins"]
7. # Plot the pie chart with calories and labels
8. plt.pie(calories, labels=labels, autopct="%1.1f%%")
9. # Add a title for the pie chart
10.plt.title("Percentages of Daily Calories Consumed from
   Carbohydrates, Fats, and Proteins")
11. # Show the pie chart
12.plt.savefig("piechart1.jpg", dpi=600, bbox_inches='tigh
   t')
plt.show()
```

Output:

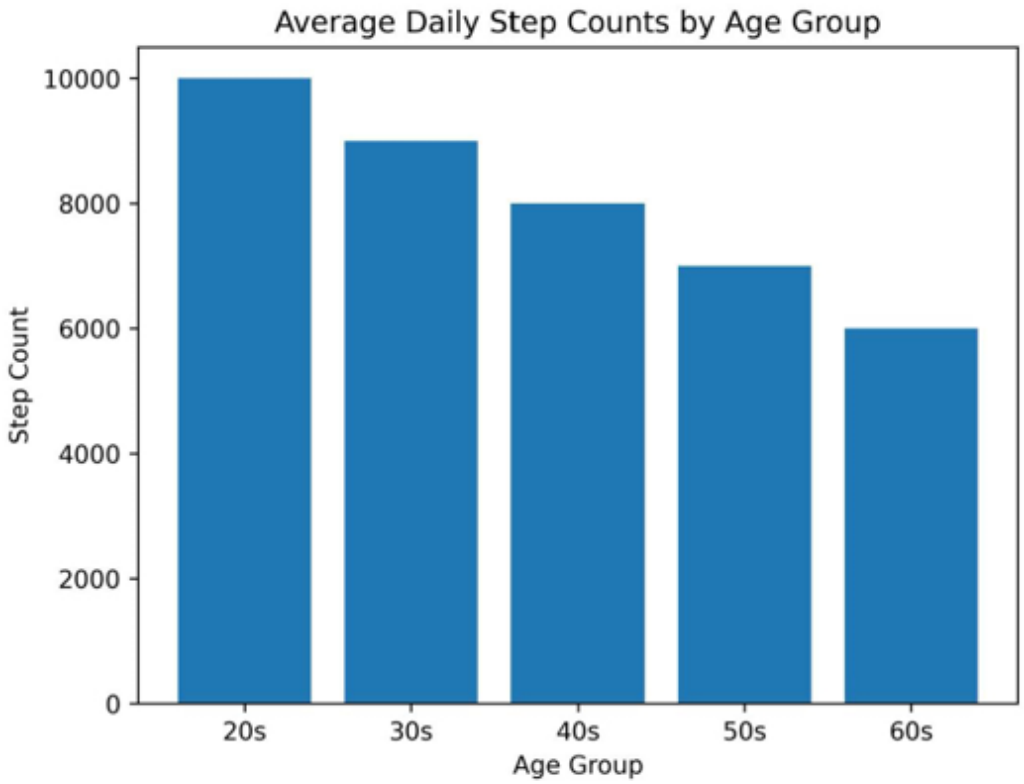

Figure 2.7: *Daily step counts of people in different age category using bar chart*

Histogram

Histograms are used to visualize the distribution of continuous data or to understand the frequency of values within a range. Mostly used for continuous data. For example, to show **Body Mass Indexes (BMIs)** in a large sample of individuals to see how the population's BMIs are distributed.

Tutorial 2.37: An example to plot distribution of individual BMIs in a histogram plot, is as follows:

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. # Generate a large sample of BMIs using numpy.random
   .normal function
4. # The mean BMI is 25 and the standard deviation is 5
5. bmis = np.random.normal(25, 5, 1000)
6. # Plot the histogram with bmis and 20 bins
7. plt.hist(bmis, bins=20)
8. # Add a title for the histogram
9. plt.title("Histogram of BMIs in a Large Sample of Individ
   uals")
10. # Add labels for the x-axis and y-axis
11. plt.xlabel("BMI")
12. plt.ylabel("Frequency")
13. # Show the histogram
14. plt.savefig('histogram.jpg', dpi=600, bbox_inches='tight'
    )
15. plt.show()
```

Output:

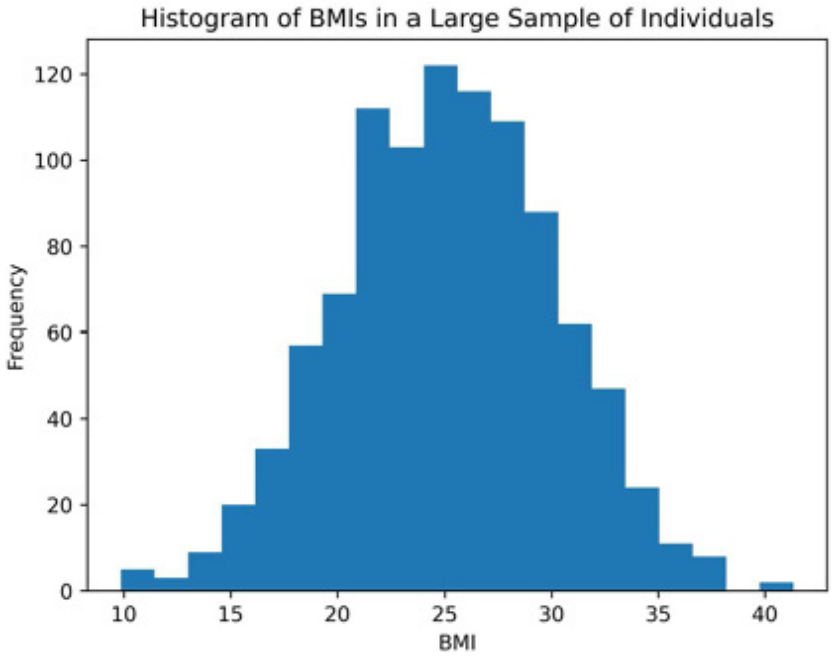

Figure 2.8: *Distribution of Body Mass Index of individuals in histogram*

Scatter plot

Scatter plots are ideal for visualizing relationships between two continuous variables. It is mostly used for two continuous variables that you want to analyze for correlation or patterns. For example, plotting the number of hours of sleep on the x-axis and the self-reported stress levels on the y-axis to see if there is a correlation between the two variables.

Tutorial 2.38: An example to plot number of hours of sleep and stress levels to show their correlation in a scatter plot, is as follows:

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. # Generate a sample of hours of sleep using numpy.rand
   om.uniform function
4. # The hours of sleep range from 4 to 10
5. sleep = np.random.uniform(4, 10, 100)
6. # Generate a sample of stress levels using numpy.rando
```

```
m.normal function
# The stress levels range from 1 to 10, with a negative correlation with sleep
stress = np.random.normal(10 - sleep, 1)
# Plot the scatter plot with sleep and stress
plt.scatter(sleep, stress)
# Add a title for the scatter plot
plt.title("Scatter Plot of Hours of Sleep and Stress Levels")
# Add labels for the x-axis and y-axis
plt.xlabel("Hours of Sleep")
plt.ylabel("Stress Level")
# Show the scatter plot
plt.savefig("scatterplot.jpg", dpi=600, bbox_inches='tight')
plt.show()
```

Output:

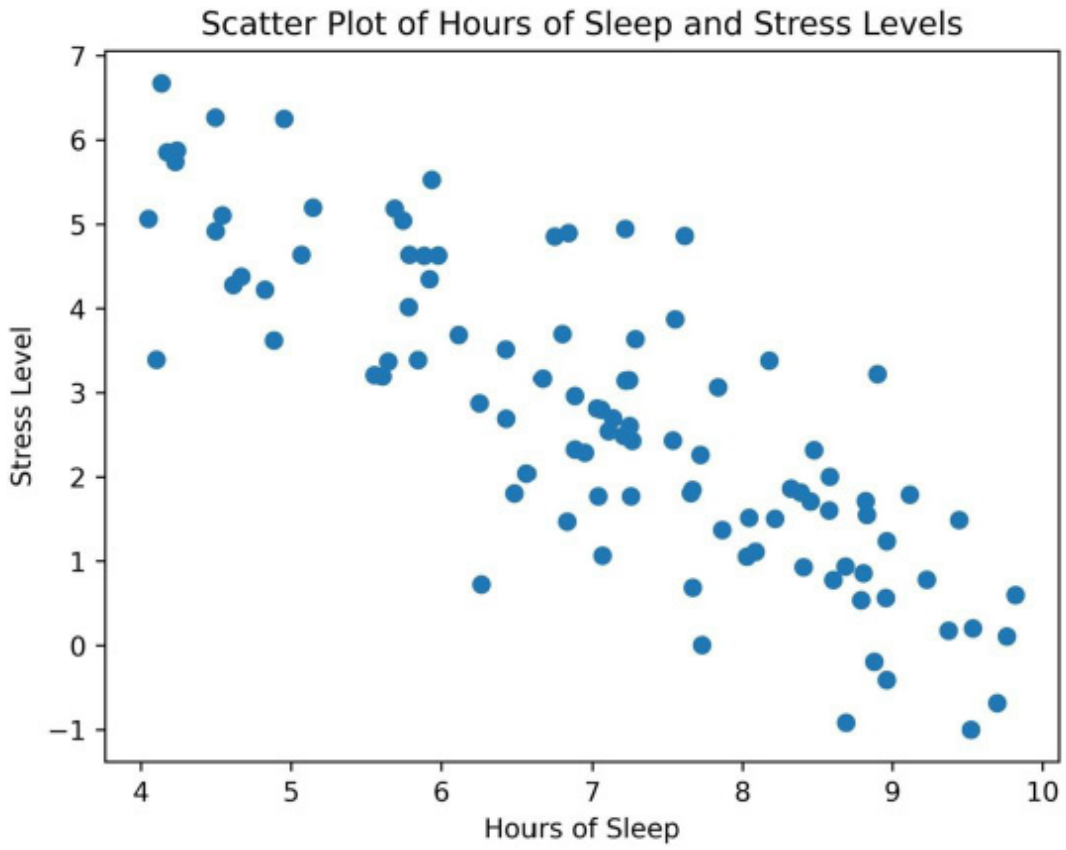

Figure 2.9: *Number of hours of sleep and stress levels in a scatter plot*

Stacked area plot

Stacked area chart illustrates the relationship between multiple variables throughout a continuous time frame. It is a useful tool for comparing the percentages or proportions of various components that comprise the entirety.

Tutorial 2.39: An example to plot patient count based on age categories (child, teen, adult, old) over the years using stacked area plot, is as follows:

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. # Create a sample data set with four variables
4. x = np.arange(2020, 2025)
5. y1 = np.random.randint(1, 10, 5)
```

```
y2 = np.random.randint(1, 10, 5)
y3 = np.random.randint(1, 10, 5)
y4 = np.random.randint(1, 10, 5)
# Plot the stacked area plot with x and y1, y2, y3, y4
plt.stackplot(x, y1, y2, y3, y4, labels=
["y1", "y2", "y3", "y4"])
# Add a title for the stacked area plot
plt.title("Stacked Area Plot of Sample Data Set")
# Add labels for the x-axis and y-axis
plt.xlabel("Year")
plt.ylabel("y")
# Add a legend for the plot
plt.legend()
# Show the stacked area plot
plt.savefig('stackedareaplot.jpg', dpi=600, bbox_inches=
'tight')
plt.show()
```

Output:

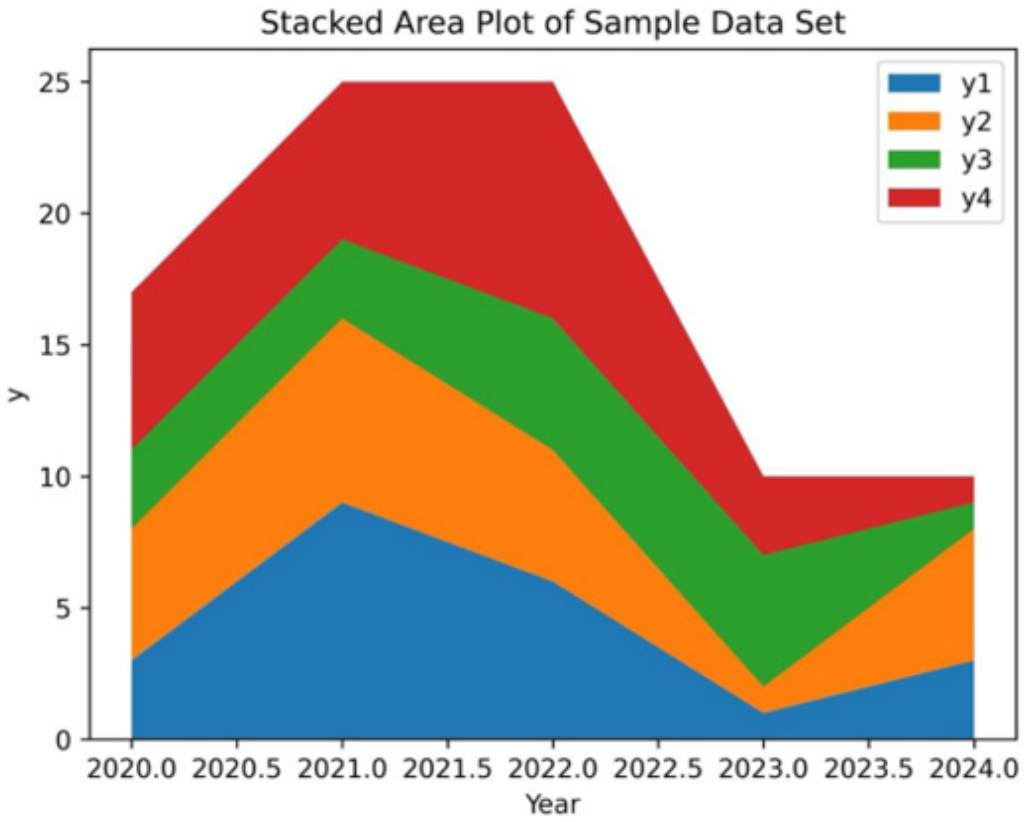

Figure 2.10: *Number of patients based on age categories in stacked area plot*

Dendrograms

Dendrogram illustrates the hierarchy of clustered data points based on their similarity or distance. It allows for exploration of data patterns and structure, as well as identification of clusters or groups of data points that are similar.

Violin plot

Violin plot shows how numerical data is distributed across different categories, allowing for comparisons of shape, spread, and outliers. This can reveal similarities or differences between categories.

Word cloud

Word cloud is a type of visualization that shows the

frequency of words in a text or a collection of texts. It is useful when you want to explore the main themes or topics of the text, or to see which words are most prominent or relevant.

Graph

Graph visually displays the relationship between two or more variables using points, lines, bars, or other shapes. It offers valuable insights into data patterns, trends, and correlations, as well as allows for the comparison of values or categories. It is suggested to use graphs for data analysis.

Conclusion

Exploratory data analysis involves several critical steps to prepare and analyze data effectively. Data is first aggregated, normalized, standardized, transformed, binned, and grouped. Missing data and outliers are detected and treated appropriately before visualization and plotting. Data encoding is also used to handle categorical variables. These preprocessing steps are essential for EDA because they improve the quality and reliability of the data and help uncover useful insights and patterns. EDA includes many steps beyond these and depends on the data, problem statement, objective, and others. To summarize the main steps, it includes. Data aggregation combins data from different sources or groups to form a summary or a new data set. Data aggregation reduces the complexity and size of the data, and to reveal patterns or trends across different categories or dimensions. Data normalization scales the numerical values of the data to a common range, such as 0 to 1 or -1 to 1. Data normalization reduces the effect of different units or scales on the data, making the data comparable and consistent. Data standardization contributes to remove the effect of outliers or extreme

values on the data, and to make the data follow a normal distribution. The data transformation helps to change the shape or distribution of the data, and to make the data more suitable for certain analyses or models. Data binning is dividing the numerical values of the data into discrete intervals or bins, such as low, medium, high, etc. Data binning can help to reduce the noise or variability of the data, and to create categorical variables from numerical variables. The data grouping groups the data based on certain criteria or attributes, such as age, gender, location, etc. Data grouping helps to segment or classify the data into meaningful categories or clusters, and to analyze the differences or similarities between groups. Data encoding techniques, such as one-hot encoding, label encoding, and ordinal encoding, convert categorical variables into numerical variables, making the data compatible with analyses or models that require numerical inputs. Data cleaning detects and treats missing data and outliers. Similarly when performing EDA of data, data visualization assists to understand the data, display the summary, view the relationship among the variables through charts, graphs and other graphical representations. As you begin your work in data science and statistics, these steps cover the things you need to consider. So, this is the initial step while working with data, and everything starts with this.

In *Chapter 3: Frequency Distribution, Central Tendency, Variability*, we will start with descriptive statistics, which will delve into ways to describe and understand the pre-processed data based on frequency distribution, central tendency, variability.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com